Towards a Scalable Infrastructure for Semantic Web Services Execution

Antonio J. Roa-Valverde, and José F. Aldana-Montes
Department of Computer Languages and Computing Sciences, University of Málaga, Spain
Email: {roa, jfam}@lcc.uma.es

***Abstract*— In this work, we propose the implementation of an infrastructure compliant with the principles established by the OASIS Semantic Execution Environment TC. We use an Enterprise Service Bus (ESB) as the backbone for our proposal.**

We believe that developed approaches to model Semantic Web Services must be put in practice. In this way it is possible to use Semantic Web Services in conjunction with an ESB to define a Semantic Enterprise Service Bus (SESB). The SESB provides mechanisms to collect all these technologies together and acts as a layer to overcome the application integration problem.

Measurements show that our platform imposes acceptable overheads when enforcing the described design.

***Index Terms*— Semantic Web, Semantic Web Services, middleware, Enterprise Service Bus, Application Integration Patterns**

I. Introduction

Years ago researchers envisaged the future Web as a Web populated by an enormous amount of information shared among users, from users to applications and even from applications to applications (the latter also known as A2A interaction). Today it is a reality. In this context, the Semantic Web tries to formalize the knowledge available among the different resources in order to facilitate the information usage. Further to the A2A interaction there is a lot of effort focusing on a more efficient and scalable solution which can address the drawbacks of dealing with this amount of resources. In this way, the term resource stands for any piece of public information available as simple data or as a data provider implemented as a service. OASIS [29] defines a service as "a mechanism to enable access to one or more capabilities, where the access is provided using a prescribed interface and is exercised consistent with constraints and policies as specified by the service description". In other words, a Web Service provides one way of implementing the automated aspects of a given business or technical service.

The Semantic Web Service Initiative[1] (SWSI) focuses on improving the life-cycle of conventional Web Services in order to extend the SOA tendencies independently of the scalability limitations. As it is established by this consortium: "... to achieve the above overall mission, a number of theoretical, methodological and empirical issues must be addressed. These include:

- creation of language and ontological infrastructure to support incorporation of machine understandable semantics into Web Services;
- development of appropriate Web Services architecture and applications."

Refering to these ideas there are several proposals that have been submitted to W3C for evaluation: OWL-S [1], WSMO [2], WSDL-S [3], SWSF [4], but not so many implementations applicable to real scenarios, which means that there is still a gap among academy and company regarding to this context. The Semantic Annotations for WSDL and XML Schema (SAWSDL) [5] is the exception. SAWSDL reached recommendation status on August 28 2007, turning it into a "W3C Standard". Nowadays, this is the most advanced way to model Semantic Web Services following a bottom-up approach.

The use of Semantic Web Services technology in enterprises would not be possible without the existence of an infrastructure that allows covering the life-cycle of Web Services using semantic annotation techniques. The OASIS Semantic Execution Environment Technical Commitee (SEE TC) [24] addresses this problem and tries to provide guidelines, justifications and implementation directions for an execution environment for Semantic Web Services.

In this work, we propose the implementation of an infrastructure compliant with the principles established by the SEE TC where an Enterprise Service Bus (ESB) [12] is the backbone. An ESB allows the cooperation and the exchange of data between heterogeneous systems. It is a logical architecture based on the principles of SOA, which aims to define services explicitly and independently of the implementation details. It also pays close attention to securing a transparent location and excellent interoperability.

We believe that developed approaches to model Semantic Web Services must be put in practice. In this way it is possible to use Semantic Web Services in conjunction with an ESB to overcome the application integration problem [10]. The objective is to define a Semantic Enterprise Service Bus (SESB), providing mechanisms to collect all these technologies together and acting as a layer to access services through the invocation paradigm based on goals, in the same way as WSMO/X does [25].

This paper is based on "Extending ESB for Semantic Web Services Understanding," by Antonio J. Roa-Valverde, and José F. Aldana-Montes, which appeared in R. Meersman, Z. Tari, and P. Herrero (Eds.): OTM 2008 Workshops, LNCS 5333, pp. 957964, 2008. ©Springer-Verlag Berlin Heidelberg 2008.

[1]http://www.swsi.org/

doi:10.4304/jsw.4.8.821-832

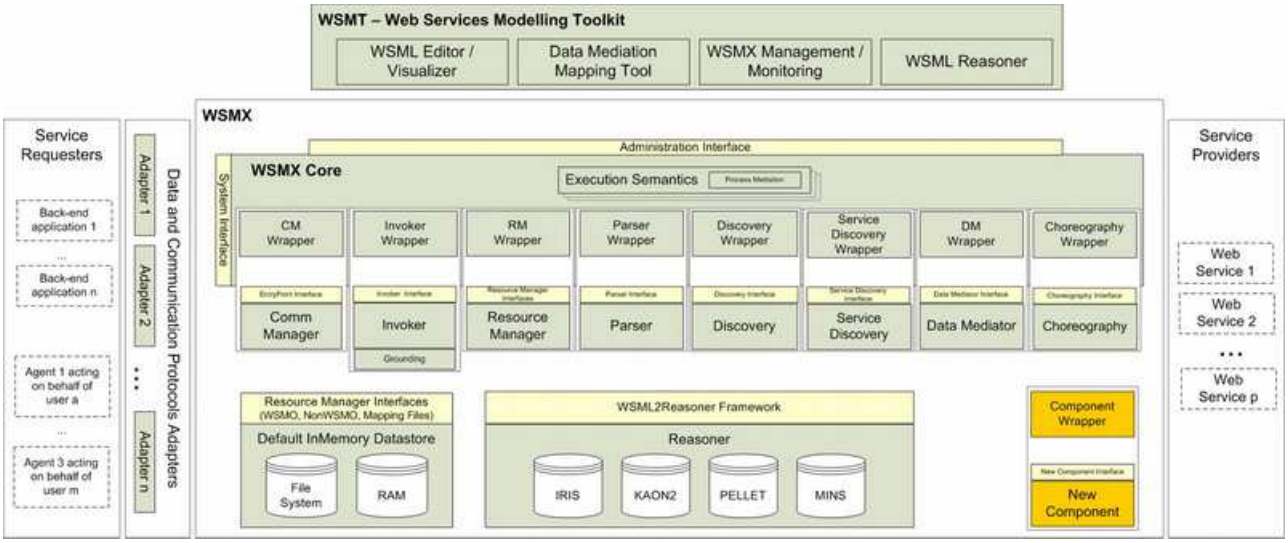

Figure 1. WSMX v.0.5 architecture overview [extracted from [18]]

This document makes reference to WSMX[2] [30] as an environment built with the aim of automating the life-cycle of Web Service's usage. WSMX is a testbed for WSMO and demonstrates the viability of using WSMO to put in practice the SWS related ideas. We provide a description of the task carried out in order to combine the WSMX functionality with the use of Enterprise Application Integration patterns [23] within the ESB.

The remainder of this paper is structured as follows. In Section II, we describe WSMX and the main concepts around ESBs. We also discuss other related work. Section III addresses the new approach and highlights its importance in the Service Oriented Computing (SOC) context. Section V shows how the platform behaves after the execution of stress tests. Finally, in Section VI conclusions and future work are summarized.

II. Background and Related Work

In this section, we describe WSMX and main concepts around ESBs to provide the context for the discussion of our contribution in Section III. We also discuss other related work.

A. Web Service Modelling eXecution environment

WSMX follows a staged component-based software development [7] as can be observed in Figure 1. From the beginning, WSMX was considered as a decoupled and extensible framework for SWS execution. Looking at it more closely it can be appreciated that the developed architecture separates the different functionalities already provided. In this way, WSMX implements three kinds of components, i.e: components that offer the functionality of each phase in the SWS life-cycle, components orchestrating other components to achieve the desired functionality and components used to interact with the user. The expected functionality of WSMX can be described in terms of the aggregated functionality of all its components.

Components included in the first group are *Discovery*, *Service Discovery*, *Invoker*, *Data Mediator* and *Choreography*. Components used to assist the functionality of other components are *Resource Manager*, *WSML2Reasoner Framework*, *Communication Manager* and also the *WSMX Core* can be classified here. Finally, components that integrate the *Web Services Modelling Toolkit* (WSMT) are included in the group related with the interaction with the user. The platform also provides the capability to add new components as soon as the requirements change. This is possible in part due to the *WSMX Core* and the use of wrappers that abstract the communication with each component.

The way how the whole platform behaves is known as *execution semantics* and it is hard-coded inside the *WSMX Core* [8], which is responsible for the interaction among the different components and coordinates the messages flow. Currently, there is a centralized version of the *WSMX Core*, however the first ideas in the development of WSMX were to offer a distributed framework in order to configure a cluster of WSMX instances deployed across a network. Nowadays this has not yet been implemented.

Refering to the execution semantics, the current implementation requires much effort if the system needs to add new functionality. In this way, the developer has to understand how the whole platform behaves reading lines and lines of code. Recent ideas in the development of WSMX try to adapt the execution semantics to a declarative style. Following this approach, in Section IV-C we show an alternative, describing how to achieve in a different and more efficient way the targets that WSMX pursued from the beginning.

1) WSMX architecture in depth: The following descriptions depict an overview of the functionality encapsulated inside each WSMX component. For more details see [7].

[2]http://www.wsmx.org/

Core: it constitutes a microkernel providing middleware functionality such as finding and looking for components, message handling and defining execution paths ("execution semantics").

Choreography: defines how to interact with the Web Service messages exchange. It also resolves process heterogeneity (in terms of communication mismatches) between service requester and provider.

Communication Manager: constitutes the entry point to the WSMX system exposing the functionality of the other components.

Data Mediator: transforms instances of the ontologies known to one of the involved parties to instances of the ontologies known to the respective other party, or viceversa, based on previously created abstract mappings between ontologies. It resolves data heterogeneity that can appear during discovery, composition, selection or invocation of Web Services.

Invoker: handles communication between WSMX and external SOAP-based Web Services. It includes lifting/lowering to/from WSML to XML.

Orchestration: resolves process heterogeneity in terms of defining how the overall functionality of a service is achieved by cooperating with other services. Currently this functionality is not implemented in WSMX, but the choreography component provides some interfaces that can be used during the composition process.

Parser: performs a syntactic validity check of WSML documents and converts it to an in-memory representation. The parser is used in the WSMX execution semantics. Actually, the parser does not constitute a component like the other modules. It represents a set of functions implemented in the WSMO4j[3] library and it is accessible at each step of the SWS life-cycle.

Resource Manager: stores all the information that WSMX uses, namely WSMO definitions (web services, ontologies, goals and mediators), non-WSMO data (events and messages) and WSDL documents used for grounding.

Discovery: enables the discovery of Web Services by finding Web Service descriptions that match the goal specified by the requester.

Service Discovery: extends the functionality of the Discovery component providing service contracting and QoS discovery (service selection based on non-functional properties).

Figure 2 depicts the dependencies among the components that will be reused and deployed on the ESB. It should be noted that components responsible for the communication and cooperation (WSMX Core, Communication Manager and Invoker) have been dropped from the architecture because the ESB already provides this functionality and can be adapted to our requirements.

2) Execution Semantics: As stated previously, the whole system behaves are modelled through the concept of execution semantics. This behaviour includes the interaction among the WSMX components to achieve the user's requirements, commonly modelled as WSMO goals. Since WSMX is an event driven system, its behaviour is specified by the order of events. Event exchange is conducted via a Tuple Space [9], which provides a persistent shared space enabling seamless interaction between components without direct event exchange between them. Interactions are carried out by exploiting a publish-subscribe mechanism. Figure 3 depicts an architectural overview of the WSMX communication model. The communication between WSMX client, WSMX and the endpoint Web Services requires all the communication parties to be subscribed to appropriate events, for a successful communication. The same happens with the communication among WSMX components. The current implementation is for the communication and coordination of components internal to WSMX only. It does not include communication between different WSMX instances. Further details about how Tuple Spaces and the Triple Space Computing (TSC) model are related to each other within WSMX can be found in [19]. The event driven approach already used allows migrating the components over an ESB easily. This deployment provides great advantages as will be outlined in the next section.

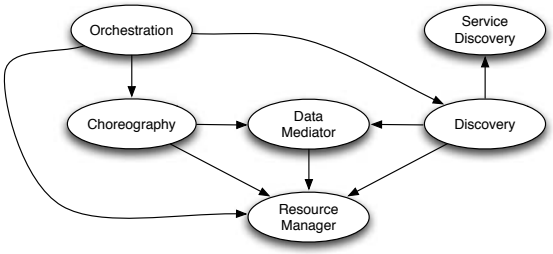

Figure 2. Dependencies among WSMX components

In the current WSMX version, there is one instance per component and there is also one possible sequential execution of the execution semantics at the same time, i.e., it can not run multiple goals in parallel. Deploying the WSMX functionality on the ESB makes it possible to manage different execution semantics concurrently, thereby overcoming the previous limitation of running only one goal in a batch process style.

The execution semantics is hard-coded inside the WSMX Core component as part of the coordination model. Despite WSMX being an extensible infrastructure a lot of effort is necessary if new components are to be plugged into the system. The addition of a new component would involve the codification of its behaviour. This can be a problem when the new component takes part in different execution semantics already implemented or if it defines new functionality that involves an independent execution semantics. In this way, adding new functionality in the form of a new component constitutes a tedious task. In order to clarify this issue, Figure 4 depicts an UML diagram representing how the execution semantics

[3]http://wsmo4j.sourceforge.net/

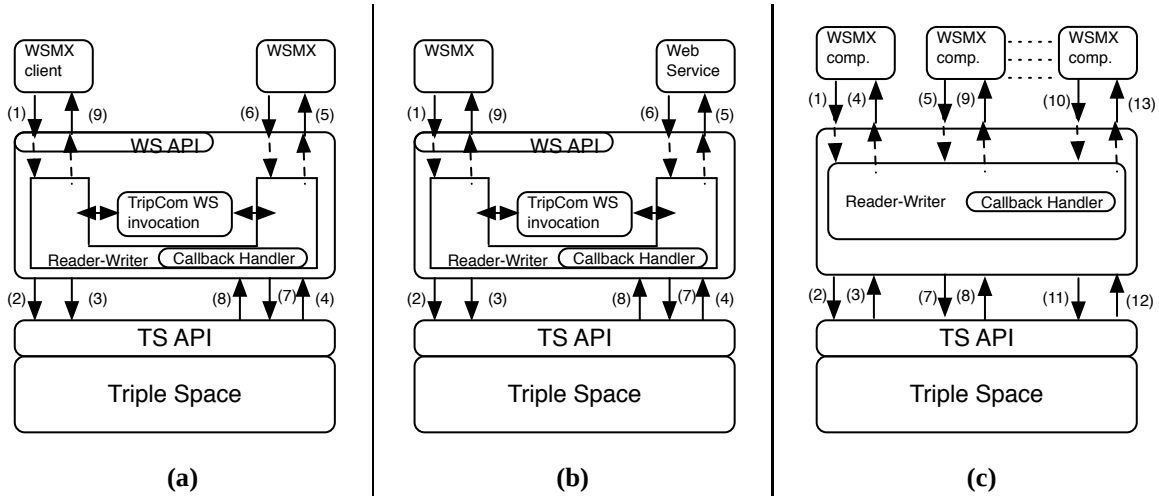

Figure 3. WSMX v.0.5 communication model (extracted from [31]): (a) Internal behavior of WSMX Client invoking WSMX via Triple Space. (b) Internal behavior of WSMX Invoker invoking Web Service via Triple Space. (c) Component Management in WSMX using Triple Space

is implemented inside the WSMX Core. Each class at the top and inheriting from *WSMXExecutionSemantic* represents the entry point to the respective execution semantics. Classes inheriting from *AchieveGoalChoreography* represents existing states within that execution semantics. The change from one state to other is controlled by the *Context* class. States from an execution semantics could make use of states from a different execution semantics. This happens when an execution semantics requires the functionality implemented in other execution semantics. For example, in Figure 4 *AchieveGoalChoreography* makes use of *DiscoverWebServices* through the state *Discovery*. For more clarity, Figure 4 depicts only states belonging to *AchieveGoalChoreography*.

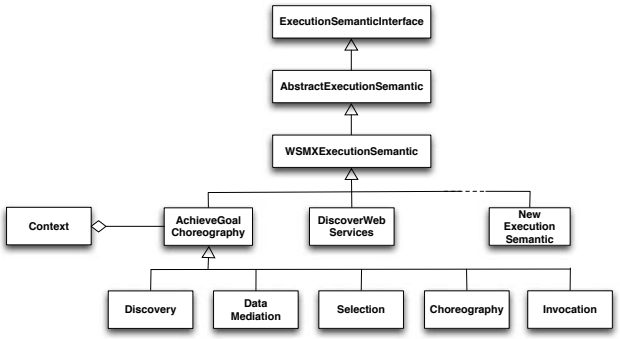

Figure 4. State design pattern used to model the execution semantics. This pattern is used in computer programming to represent the state of an object. It is a clean way for an object to partially change its type at runtime

A good approach to fix the drawbacks related with the extension of the platform would be possible with an architecture allowing the separation of deployment and communication processes. This is what an ESB facilitates. On the one hand an ESB is built on top of a layer that allows the addition of new components as if they were plugins extending a software packet. On the other hand, the backbone of an ESB is constituted by a communication system that enables the interaction among the applications deployed on top of the ESB. With these features it makes sense to apply the ESB functionality to extend the WSMX platform because it can be reduced to an integration problem where the applications to be integrated are the different WSMX components.

B. Enterprise Service Bus

From the beginning, WSMX was conceived as an infrastructure to demonstrate the viability of WSMO. In this way, the different tasks carried out during the development process have focused more on implementing the functionality related with the SWS life-cycle. Nowadays, the necessity for a more scalable infrastructure has forced WSMX to redefine its initial design.

The next versions of WSMX should take into account a distributed architecture allowing the addition of new functionality any time it is so required by the system. The new requirements should not change the already implemented infrastructure. In this way, it is necessary to put in practice some principles related to application integration techniques [10]. To do this we rely on the ideas provided by an Enterprise Service Bus (ESB) [11].

Basically, an ESB constitutes a middleware for Enterprise Application Integration (EAI). An ESB makes Web Services, XML, and other integration technologies immediately usable with the mature technology that exists today. The core tenets of SOA are vital to the success of a pervasive integration project, and are already implemented quite thoroughly in the ESB. The Web Service standards are heading in the right direction, but remain incomplete with respect to the enterprise-grade capabilities such as security, reliability, transaction management, and business process orchestration. The ESB is based on established standards in these areas, and has real implementations that are already being deployed across a number of industries. The ESB is capable of keeping in step with the ongoing evolution of the Web Services equivalents of these capabilities as they mature [12]. It would be interesting to maintain these capabilities using Semantic Web Services.

The ESB will replace the current *WSMX Core* functionality providing an abstraction layer responsible for the communication and integration of new components in the platform. In order to build a loosely decoupled infrastructure we will rely on an asynchronous message oriented middleware (MOM) namely ActiveMQ[4], wich supports the Java Message Service (JMS) specification [13]. Tasks concerned with integration matters will be facilitated using the Java Business Integration (JBI) specification [14].

There are several options from different organizations and companies supporting the concept of an ESB [15]. This makes the choice of a specific implementation difficult. We have chosen Apache ServiceMix[5] as the ESB implementation. ServiceMix provides an open source approach supporting the SOA principles using specifications such as JBI, JMS, JMX [16] and OSGi [17] among others. In our analysis we pay special close attention to the choice of a portable and scalable platform[6]. These issues draw our attention to ESBs compliant with the JBI specification. A plausible justification for this is that JBI components developed for a particular solution can be used in any other JBI compliant ESB.

C. Related Work

The main challenge among researchers in the Semantic Web Service field lies in overcoming the technological gap between the use of syntactic technology and semantic technology. As we can see in [10], many R&D projects are ongoing with the aim of bringing semantics into SOA. During the last five years some attempts have been made to implement platforms to achieve the Semantic Web Services challenge[7], most of this work has been part of EU funded research.

Infrawebs[8] was the first collaboration between academia and industry addressing the search of such a platform. In this project, the WSMX functionality was distributed among different components and deployed like Web Services on top of an ESB, namely, Mule ESB[9]. The main target of this platform involved developing a solution to lead the application of Semantic Web Services to real scenarios. Nevertheless, because the system was designed without taking into account the principles stated in this document all the usage forecasts proved to be wrong.

Most recent projects in this area pursue a similar approach to our ideas. SOA4All[10] is an ongoing project that tries to develop an infrastructure compatible with the next generation of the Web, where billions of services will be shared and used by consumers. The envisaged infrastructure relies on the use of a JBI-compliant ESB which will be extended using the TupleSpace Computing (TSC) [9] approach for the communication among services. As was established in [21], "the outcome of the project will be a comprehensive framework and infrastructure that integrates four complimentary paradigm-shifting technical advances into a coherent and domain independent service delivery platform: Web principles and technology..., Web 2.0..., Semantic Web... and context management..." Although this project defends the same interests, the platform described in this work follows a different approach for the communication process. While SOA4All makes use of TSC, we opt for the use of Enterprise Application Integration (EAI) patterns [23]. The comparison of these two approaches will be the main focus in a future analysis.

There is another ongoing project known as Semeuse[11] which is closer to our ideas. Semeuse tries to extend the role of ESBs using semantic technologies, its main focus being the application to service composition tasks. Similarly to SOA4All, Semeuse relies on a JBI-compliant ESB, namely, PEtALS ESB[12]. This issue makes possible a future collaboration and reuse of the ideas presented in these projects and our approach, since we are following the JBI specification [14].

Current efforts show that researchers have become aware of the actual technological transition in SOA. In this sense, it is difficult to know when Semantic Web Services may be used among ICT enterprises without any limitation. For the moment, researchers should postpone the development of new platforms that cover the SWS life-cycle focusing their effort on obtaining a solution to overcome the current transition problem between SOA and Semantic SOA. This last issue has contributed to the development of this work.

III. Motivation

For several years many approaches to overcome the application integration problem have been proposed, i.e. CORBA [26], EAI [23], ESB [12], etc. Despite these approaches relying on different technologies and mechanisms, they share a common point of view: software engineers are responsible for understanding the different application specifications and coordinating them to build a more complex system. Figure 5 (a) depicts the necessary process to deploy a solution using an ESB. This process consists of two phases. Firstly, the software engineer must create the configuration file used for the ESB to initialize listeners in the startup phase. In this way, the software engineer must know with a high level of detail the different applications that he/she wants to integrate, i.e. accepted inputs and outputs, listener ports, protocols, etc. Secondly, in the execution phase the ESB is ready to accept messages and transport them among applications using the information stored in the configuration file. As

[4] http://activemq.apache.org/

[5] http://servicemix.apache.org/

[6] For more information about how to evaluate and choose the best solution according to the requirements the following website provides a good guideline http://servicemix.apache.org/how-to-evaluate-an-esb.html/

[7] http://sws-challenge.org/

[8] http://www.infrawebs.org/

[9] http://www.mulesource.org/

[10] http://www.soa4all.eu/

[11] http://www.semeuse.org/

[12] http://petals.ow2.org/

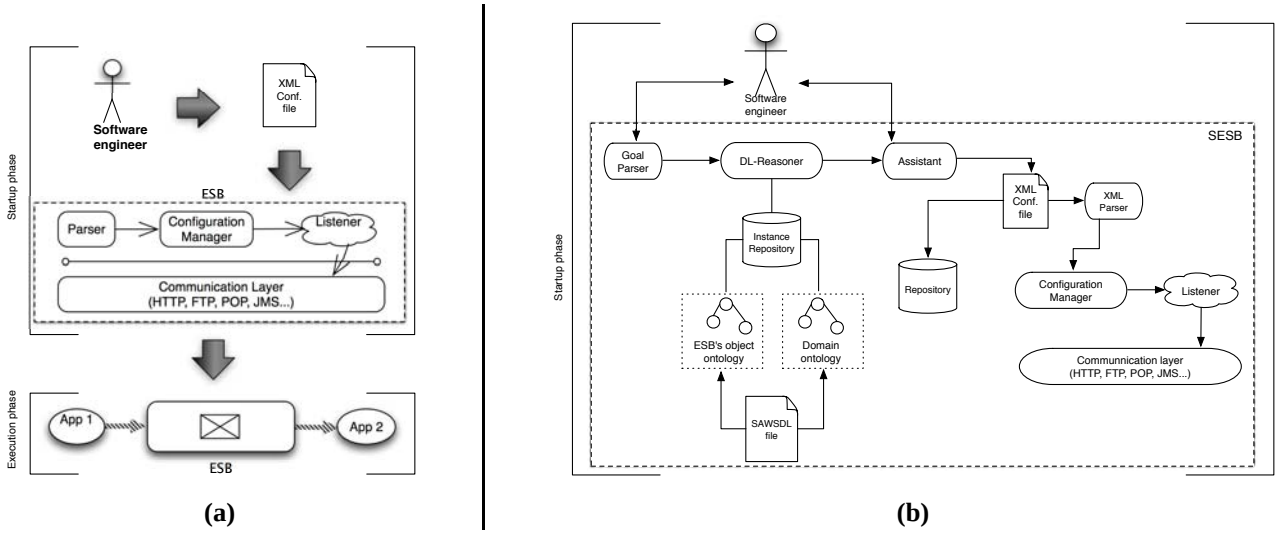

Figure 5. (a) Typical ESB usage. (b) SESB user interaction model.

we can see, the entire process relies on the configuration file coded manually by the software engineer.

Until today, proposals have been focused on providing a middleware to solve heterogeneity and communication problems among applications without taking into account information relative to the meaning of the data that these applications can process. So, a tool capable of processing this kind of information would be very helpful for software engineers. Our aim relies on applying this idea to Semantic Web Services. In this way, a tool like this could facilitate frequent tasks in this field such as service composition [28] and discovery [27]. This idea tries to avoid writing the configuration file manually. We can imagine a software engineer trying to integrate several Semantic Web Services with the aim of building a more complex service in a composition process. Ideally, the software engineer could introduce the required goal[13] and the ESB would be able to create the configuration file in an automatic or semi-automatic way using the available semantic annotations (see Figure 5 (b)).

The SESB aims at providing developers with a middleware that facilitates application integration tasks through Semantic Web Service technology. There are two different ways to build such infrastructure using an ESB. The first one uses the ESB as the base layer for building the architecture on which different components are deployed. Those components will be responsible for the manipulation of the semantic required by Semantic Web Services. In this way, the ESB does not realize the existence of semantic information and treats those components as usual. The second one tries to extend the ESB adding a new module responsible for understanding the semantic annotations over the artifacts deployed on the ESB.

In this work, we combine both approaches. On the first hand, we take advantage of the WSMX functionality applying the first approach. WSMX components will provide us with the required functionality to deal with Semantic Web Services. However, deploying each WSMX component within the ESB is not enough to manage the semantic of the information that flows encapsulated within the messages interchanged through the ESB. So, secondly, we extend the ESB with the capability of inspecting the semantic of the information available in the messages.

The main effort of this work has focused on developing the tasks included in the first approach. Nevertheless, an overview describing the functionality mentioned in the second approach has been stated.

IV. Contribution

In this section, we first show how our proposal can be used by a software engineer to perform application integration tasks. We then focus on implementation details and describe how to combine WSMX with an ESB to achieve the previous stated functionality.

A. Semantic ESB overview

The SESB deals with semantics relying on a couple of assumptions: (1) it uses available information stored as instances of an OWL-DL ontology, namely the ESB's object ontology, which models the objects that the SESB can understand (filters, transports, endpoints, etc.); (2) Web Services are annotated using SAWSDL references that point to concepts in the ESB's object ontology and concepts in a domain ontology.

The startup phase begins when the user (the software engineer) introduces the required goal. Goals represent the user's preferences and they will be introduced to the system using WSML. Therefore the developed infrastructure will be WSMO compliant. After that, a parser processes the goal and sends the information to the reasoner. This component relies on the ESB's object ontology and domain ontology to get information about suitable Web Services. The system generates a first version of the configuration file using the information provided by the reasoner. In this way, the user does not have to know low level details about Web Services, i.e. binding information.

[13]Please, see [8] for more details about the relevance of goals in execution semantics

The SESB will be able to check the compatibility between different Web Services and ask the user for required code such as the creation of adapters to overcome the heterogeneity of inputs and outputs. The assistant is the component responsible for providing this functionality and completing the configuration file. This file can be stored in a repository for later use. When the configuration file is completed the configuration manager processes it and prepares the system to receive messages in the execution phase. The figure 5 (b) depicts the aforementioned functionality.

B. Deploying WSMX components on ServiceMix

At this point, we have already mentioned what should be the purpose of a new WSMX version, namely an architecture addressing issues such as scalability and pluggability in a flexible way. In previous sections we justified the use of an ESB as an appropiate infrastructure to achieve our targets. In this section, we explain how to deploy the WSMX components on ServiceMix. In order to understand the deployment process the reader needs to be familiar with the JBI specification (see [14]).

Components deployed on top of a JBI compliant ESB communicate internally by exchanging messages through the Normalized Message Router (NMR), which is responsible for the message normalization. ServiceMix, as a JBI compliant ESB, recognizes two kinds of components depending on the functionality, namely, *service engine* (SE) and *binding component* (BC). The service engine provides business logics to other components. On the other hand, the binding component provides connectivity to external services in a JBI environment.

Using the functionality provided by both types of components, the JBI specification allows two different ways of deploying the WSMX components:

Case A: Deploying each WSMX component as a Web Service separately from the ESB. In this case it is necessary to use an application server which is responsible for executing the business logic offered by the Web Service. Therefore, a Web Service environment like Apache Axis[14] is required. The ESB only plays the role of a communication and orchestration manager, processing the different messages to and from the Web Service that wraps the WSMX component. The application server manages all the events (exceptions, invocation methods, etc) produced at runtime. Figure 6 (a) depicts a schema that represents this scenario.

Case B: Deploying each WSMX component as a SE in the ESB. In this case, the ESB provides the container for the execution of the business logic. Communication with an external service container is not required. The SE provides the functionality for the deployment of the WSMX component. Figure 6 (b) depicts a schema that represents this scenario.

A priori, it seems that the second approach offers better performance because it minimizes the number of messages exchanged in the system. In order to justify this hypothesis we provide an analysis in Section V. The obtained results confirm that the strategy depicted in the case B is more efficient. The explanation for this relies on the number of transformations that the NMR needs to process in each scenario. In this way, the analysis shows that the number of transformations carried out by the NMR is directly proportional to the number of hops in the architecture. A hop is defined as a message exchange between a service provider and a service consumer. The case A involves 4 hops whereas the case B only 3. Note that there are at least 2 hops in each communication process between 2 components because the NMR always takes part in the message exchange.

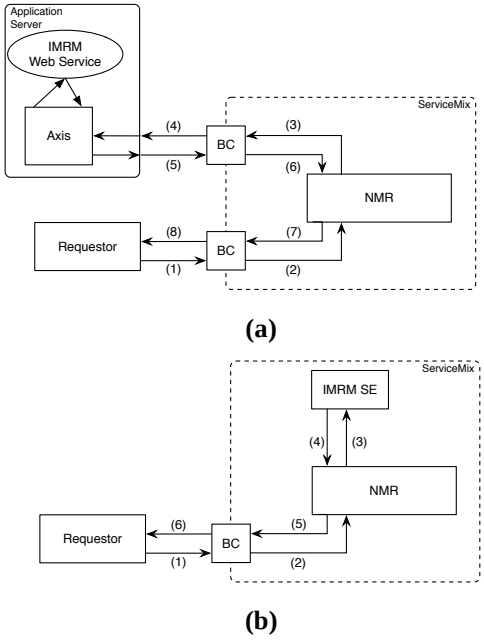

Figure 6. Possible deployment schemas in the JBI specification. (a) Deployment schema for the In Memory Resource Manager (IMRM) component wrapped as an independent Web Service using Axis2. (b) Deployment schema for the IMRM as a Service Engine within the JBI compliant ESB. For more details about the Normalized Message Router (NMR), Binding Component (BC) and Service Engine (SE), please refer to [14].

Figure 7 depicts a conceptual view of the WSMX deployment using the ESB.

C. Routing messages with the ESB: towards a declarative execution semantics

In the previous section we described the mechanism used from the JBI specification to achieve the communication process between two components. Despite the message exchange relying on this functionality, a higher level abstraction is required in order to model the execution semantics and facilitate the development.

In Section II-A.2 we compared the deployment task of the different WSMX components with an application integration problem. At this point, the following question arises, why not deploy the WSMX components inside the ESB using application integration patterns? What we need in our platform is an abstraction that allow us to

[14]http://ws.apache.org/axis2/

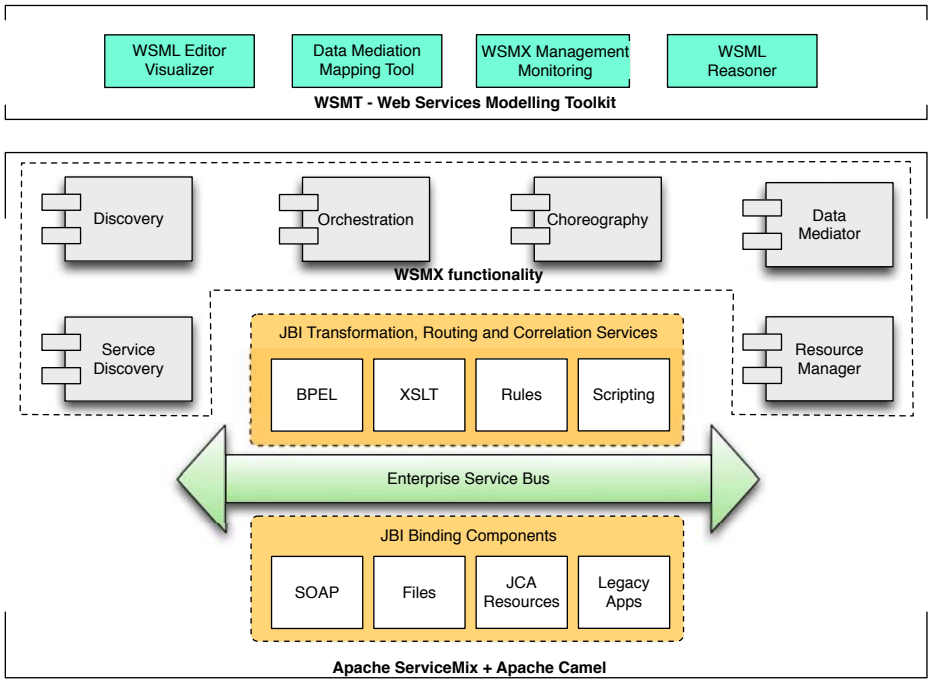

Figure 7. WSMX deployment on Apache ServiceMix

model the behaviour of the system without modifying the already implemented functionality. In this way, a software like Apache Camel[15] can be useful for this scenario. Apache Camel is an open source integration framework that implements most of the known integration patterns [23]. Apache Camel and Apache ServiceMix are fully compatible solutions appropiate for integration scenarios. The idea consists of decoupling the deployment process from the behaviour model using the integration patterns provided by Camel on top of ServiceMix.

Apache Camel provides a Java DSL (Domain Specific Language) that facilitates the implementation of the execution semantics using a declarative approach. The information codified in Java DSL is mapped to an XML file that ServiceMix is able to process. In this way, each Java DSL file will model an independent execution semantic. This allows the modification and the addition of new behaviour in the system overcoming the limitation of the previous WSMX version: there is no necessity to re-implement functionality already workings.

The first steps towards the migration of the current execution semantics model analyse the current WSMX functionality from the client point of view. The current version of WSMX exposes all the functionality like Web methods callable following a Web Service approach.

A common issue in the execution semantics is the message flow. There is no execution semantics without message exchange. Usually, when the messages flow in the system the coordinator component takes some information available in the message into account to send it to the destination. In the current WSMX version this functionality is hard-coded independently of any execution semantic model. Each time when an event comes up a state transition takes place in the system. As mentioned in section II-A.2, WSMX models the execution semantics using a state pattern [20]. States are implemented like independent objects and transitions among two states produce the change of context from the first object (previous state) to the second object (current state). These transitions are carried out through the dynamic binding mechanism implemented by object-oriented programming languages. Dynamic binding is a consequence of polymorphism. Polymorphism is the ability of one type A to appear as and be used like another type B. The main use of polymorphism is the ability of objects belonging to different types to respond to method, field, or property calls of the same name, each one according to an appropriate type-specific behavior. It is not necessary to know the exact type of the object in advance, and so the exact behavior is determined at run time (this is called dynamic binding).

Taking advantage of the information available in the messages it is possible to apply the content-based router pattern to this scenario. Basically, what this pattern can do is sending the messages to different destinations depending on the content (in the same way than a postman puts the letters in the mailbox). Figure 8 depicts the path followed by each WSMX execution semantics. In this sequence diagram, message number three is the message responsible for the initiation of the specific execution semantics by calling the first state in each specific model. Figure 9 summarizes this behaviour.

Figure 10 depicts a possible message flow implementation using EAI patterns. When the user introduces the goal a new message comes to the system through a queue

[15]http://camel.apache.org/

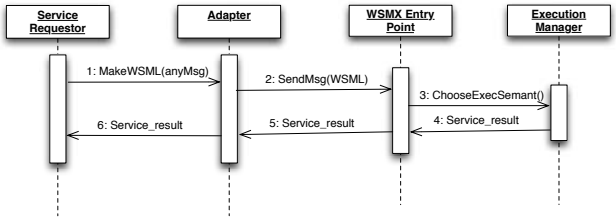

Figure 8. General entry point selection path (extracted from [8])

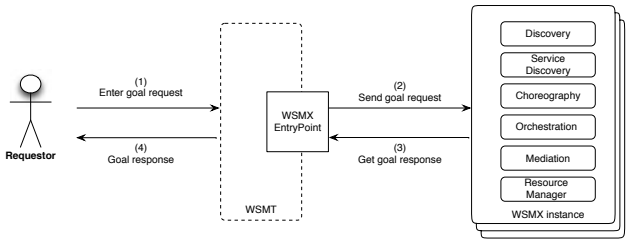

Figure 9. Global message flow overview interacting with WSMX

(*Goal.in* in Figure 10). Once the message is dispatched a correlation ID is added to it. This ID allows the identification of messages concerning a same goal. In this way, the messages exchanged by the different WSMX components, to achieve the requested goal, have the same ID during the whole execution. This mechanism allows the existence of several goals within the system in a concurrent way. The content-based router allows to consume messages from an input, evaluate some predicates and then choose the right output destination. When a message reaches its destination, i.e. the requested WSMX component, required operations are performed on it. During the execution of the operations the component can require the functionality of a different WSMX component. If this happens new messages with information about the next destination are delivered. A pipeline is needed to transform the received *in-only* message to an *in-out* message. These new messages are queued while the system is able to process them (*ES.queue* in Figure 10). The same process is repeated until the execution of the goal is done. At this point, the content-based router sends a message with the results to a queue (the *Goal.out* in Figure 10) where the requester will take it. Execution semantics is defined by the described message exchange. Predicates codified within the content-based router and the routing information generated by different WSMX components establish the message flow, i.e. the execution semantics. Note that details about synchronization regarding to the behaviour of the individual components are not discussed here, however, they need to be considered. Component behaviour and interfaces are described in WSMX architecture document [18].

The use of enterprise integration patterns allows us to replace the previous hard-coded approach of the execution semantics with a declarative approach relying on rules. This new way provides a more efficient and scalable solution that improves the previous version in the search for an appropiate environment for Semantic Web Services.

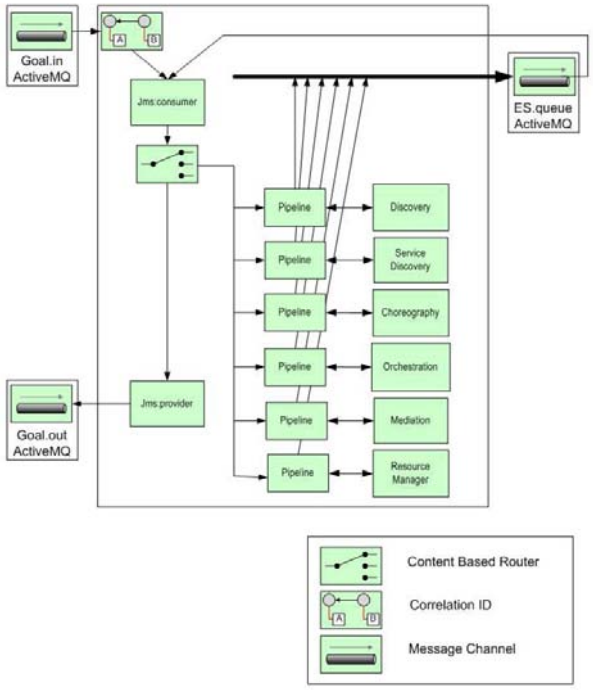

Figure 10. Message flow implementation using EAI patterns

V. Performance Tests

This analysis shows the results obtained with the test executed on the architecture in construction. The target of these tests is to weigh up the differences between the two possible approaches of deployment, namely, as an SE within the ESB (see Figure 6 (b)) or as a Web Service running in an independent application server (see Figure 6 (a)).

We have used JMeter[16] to perform the analysis. JMeter is an open source tool developed in Java that allows us to execute stress tests simulating a heavy concurrent load on a server, network or object to measure its strength or to analyze overall performance under different load types.

For our analysis we have configured a workbench using a group of 10 threads which simulates the number of users or connections to our server. The test has been executed 100 times, so in total we can study how the server behaves over a set of 1000 samples. A sample is defined as a request to the server, in our case a SOAP request. The time that JMeter requires to get all threads running is known as ramp-up period. For example, if there are 10 threads and a ramp-up period of 60 seconds, then each successive thread will be delayed by 6 seconds. In 60 seconds, all threads would be up and running. In our experiment the ramp-up period has been established at 1 second. With this test we will measure the number of requests per time (throughput) that the server is able to handle in order to achieve the invoked operation. In this way, a higher throughput means that the server requires less time to perform the request. This calculation includes any delays

[16]http://jakarta.apache.org/jmeter/

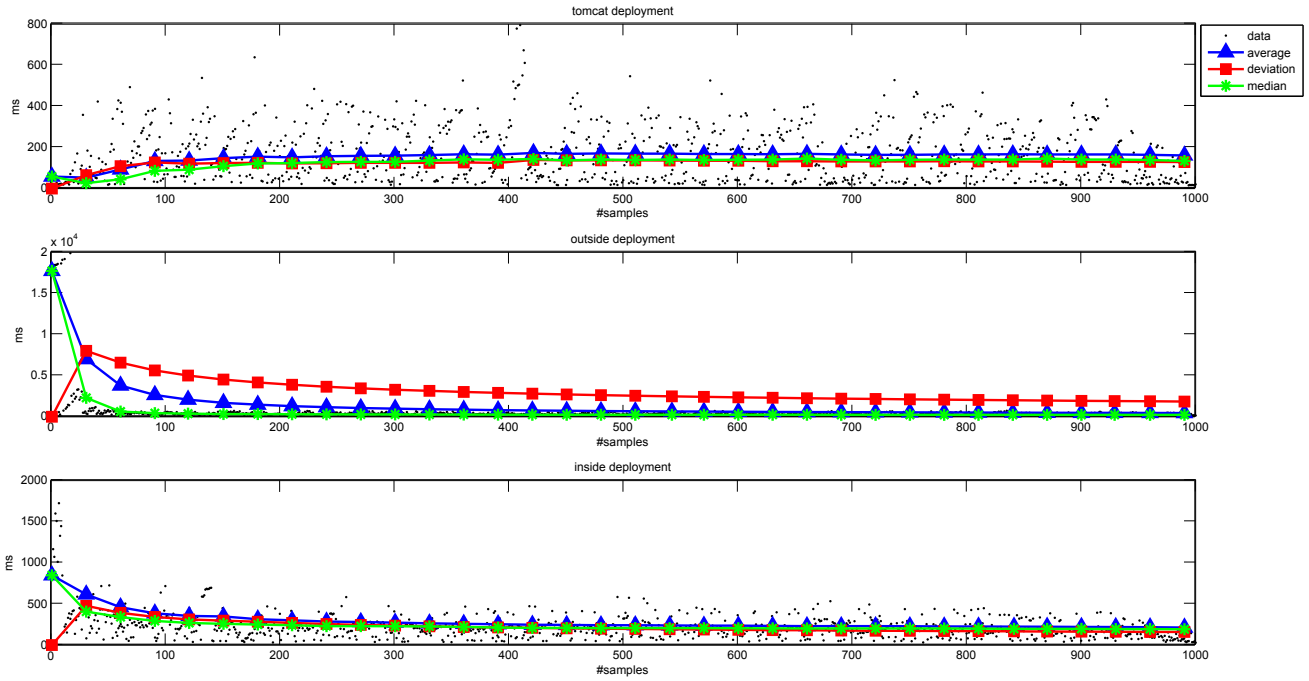

Figure 11. Test results over the different deployment strategies. From the top to the bottom: (a) deployment as an external Web Service using Apache Axis 2 and Apache Tomcat. (b) deployment on ServiceMix using a Binding Component to communicate with an external Web Service running on Apache Axis 2 and Apache Tomcat. (c) deployment as a Service Engine within Apache ServiceMix.

added to the test and JMeter's own internal processing time. The advantage of doing the calculation like this is that the result represents something real - the server in fact handled that many requests per minute, and the number of threads can be increased and/or the delays can be decreased to discover the server's maximum throughput. Whereas if calculations were made that factored out delays and JMeter's processing, it would be unclear what could be concluded from that number.

All tests have been executed in the same machine. In this way, the obtained results are independent of network issues. The machine used is a 2.2 GHz Intel Core 2 Duo with 1GB 667 MHz DDR2 SDRAM and Mac OS X v.10.5.2.

Figure 11 depicts the different scenarios. For all of them, we have performed the test against the WSMX In Memory Resource Manager (IMRM) deployed as a Web Service. From the top to the bottom: (a) deployment as an external Web Service using Apache Axis 2 and Apache Tomcat; (b) deployment on ServiceMix using a Binding Component to communicate with an external Web Service running on Apache Axis 2 and Apache Tomcat; and (c) deployment as a Service Engine within Apache ServiceMix. For each case, this figure shows how much time it takes the server to handle each request. The x-axis depicts the samples and the y-axis depicts the time in milliseconds that each sample needs to be executed. Furthermore, information about the average, median and deviation are depicted. The data legend shows us the widely dispersed data, representing the large value of the deviation across all samples for this test. In the case where the results are highly skewed or not symmetrical using "mean" would result in inaccurate representation of response time. The median value would closely approximate the response time. Comparing the three graphs, the case (a) offers the best result, while using the case (b) the worst performance is obtained. The case (c) offers an intermediate result. Note that the case (a) emulates a direct communication with the IMRM component, so it is normal that it gets the best result. The important thing here is what we can conclude from the use of an ESB regarding to graphs (b) and (c). Using an ESB increases the time of response. This fact affects the case (b) in major grade than the case (c). The explanation for this relies on the amount of messages required by each case (see Figure 6).

One more detail on these graphs can be appreciated at the beginning of each test. In case (a) the time of response increases during the first 100 samples. At this point of the execution the time of response reaches the stability. This behaviour is consequence of the stated ramp-up period which indicates that at this point all connections are sending requests to the server. On the other hand, in cases (b) and (c) the time of response decreases during the beginning of the test. This does not mean that using an ESB the system is independent of the amount of connections, but the ESB consumes more resources and it needs more time during the initialization. This time is higher in case (b) than in case (c) as can be observed in Figure 11.

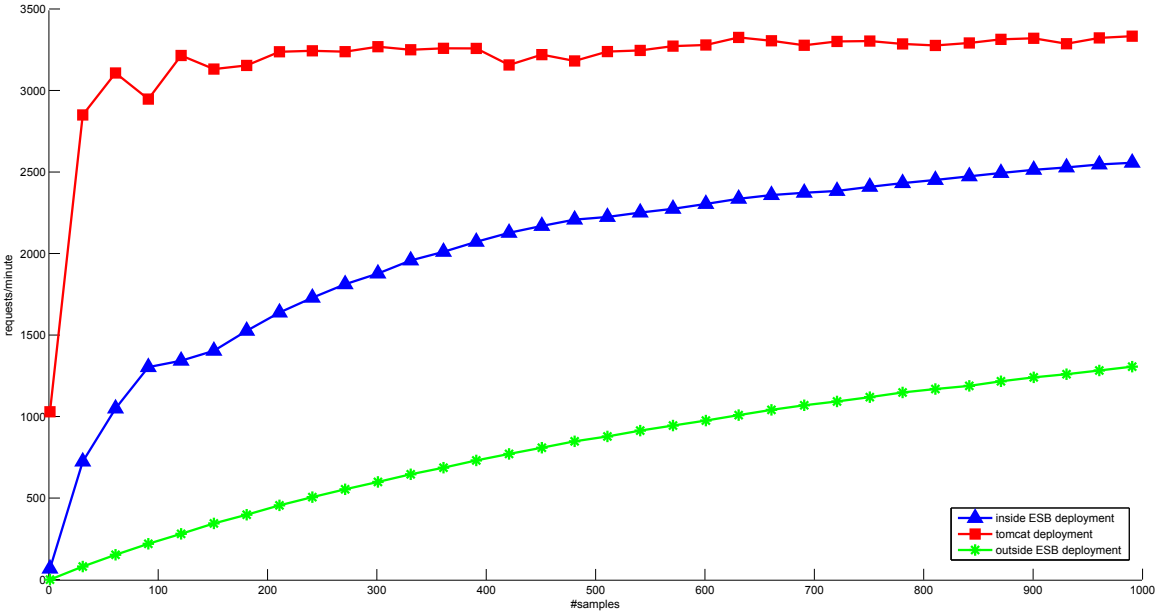

Figure 12. Throughput for each deployment strategy.

Figure 12 depicts the throughput for each scenario obtained after the execution of this test. The throughput is defined by

$$throughput = \frac{60 * 10^3}{average} requests/min$$

The results for each deployment strategy show that the throughput depends on the number of messages interchanged to handle a user request. In this way, the test demonstrates that deploying components within the ESB using a Service Engine results in a higher performance than using a Binding Component that communicates with an external application server. Our work relies on the former, while other related works as Infrawebs makes use of the latter (see Section II-C for more details related to Infrawebs).

Finally, if we compare the throughput of the latter scenarios with the throughput obtained using Apache Tomcat directly it is possible to measure the overload introduced by the ESB. As was stated previously, this overload is consequence of (1) a higher amount of message exchanges and (2) a major requirement of resources in terms of memory.

VI. Conclusions and Future work

In this paper, we describe how to build a Semantic Enterprise Service Bus combining several technologies such as OWL, SAWSDL and SOA. This kind of tool allows software engineers to apply a bottom-up design to deploy a solution that relying on the Semantic Web Services approach. The aim is to develop a platform to overcome the problems of current SOA, i.e. finding the most suitable service for a certain requirement among thousands of different services or building a complex service from other simple services. We have also described some ideas in order to improve the current WSMX version towards a more adaptive infrastructure for the envisaged Web of Services[17].

All the necessary WSMX components have been deployed as SEs within the ESB. Currently, we are improving the communication system using EAI patterns implemented by Apache Camel. In this way, our ongoing work focusses on extending the language provided by Camel with the aim of facilitating the adaptation of the platform to future changes. This issue is compatible with the idea of implementing a declarative approach for the design of the execution semantics.

As future work we plan to validate the platform using a real use case. We propose the development of adapters or wrappers over existing SOA applications as an extension to the described work. These adapters will allow the application of a semantic layer over implemented Web Services which will be reusable in the proposed SESB. In this way, we are implementing a semi-automatic tool to annotate Web Services using SAWSDL over concepts in a domain ontology. This tool will be incorporated into the SESB to facilitate the deployment of non-annotated Web Services. Preliminary results have been published in [22].

Future works will also address the extension of the functionality provided by the choreography and orchestration modules in WSMO and the implementation of sessions to facilitate government tasks during the execution stage.

[17]Visit http://www.serviceweb30.eu for more information about a future service world

ACKNOWLEDGMENT

This work is supported by grants P07-TIC-02978 (Andalusian Government), TIN2008-04844 and HA2008-0013 (Spanish Ministry of Education and Science).

We thank Srdjan Komazec of STI Innsbruck for numerous discussions concerning this work and the reviewers for their detailed comments.

REFERENCES

[1] The OWL Services Coalition. OWL-S 1.1 Release (2004). http://www.w3.org/Submission/2004/SUBM-OWL-S-20041122/.
[2] Fensel, D., Lausen, H., Polleres, A., Bruijn, J. de, Stollberg, M., Roman, D., Domingue, J.: Enabling Semantic Web Services. The Web Service Modeling Ontology. Springer, 2006.
[3] WSDL-S. http://www.w3.org/Submission/WSDL-S/.
[4] SWSF. http://www.w3.org/Submission/2005/07/.
[5] Farrell, J., Lausen, H. (eds.): Semantic Annotations for WSDL and XML Schema. W3C Recommendation (August 2007). http://www.w3.org/2002/ws/sawsdl/.
[6] OWL Web Ontology Language Overview. W3C Recommendation 10 February 2004. http://www.w3.org/TR/owl-features/.
[7] Emilia Cimpian, Matthew Moran, Eyal Oren, Tomas Vitvar and Michal Zaremba. D13.0. Overview and scope of WSMX. Technical report, February 2005. http://www.wsmo.org/TR/d13/d13.0/v0.3/.
[8] Maciej Zaremba and Eyal Oren. D13.2v0.2 WSMX Execution Semantics. Technical Report, February 2005. http://www.wsmo.org/2005/d13/d13.2/v0.2/.
[9] Lyndon J. B. Nixon, Elena Simperl, Reto Krummenacher and Francisco Martín-Recuerda. Tuplespace-based computing for the Semantic Web: a survey of the state-of-the-art. The Knowledge Engineering Review (2008), 23:181-212 Cambridge University Press.
[10] Antonio J. Roa-Valverde, Ismael Navas-Delgado, José F. Aldana-Montes. Semantic Web Services: towards an appropriate solution to application integration. Handbook of Research on Social Dimensions of Semantic Technologies and Web Services. A book edited by M. Manuela Cunha, Eva F. Oliveira, Antonio J. Tavares and Luis G. Ferreira. 2008.
[11] Antonio J. Roa-Valverde and José F. Aldana-Montes. Extending ESB for Semantic Web Services Understanding. On the Move to Meaningful Internet Systems: OTM 2008 Workshops en Robert Meersman and Zahir Tari and Pilar Herrero. LNCS 5333. Pp 957-964. ISBN 978-3-540-88874-1.
[12] David Chappell. Enterprise Service Bus. O'Reilly (2004).
[13] Java Message Service (JMS). http://java.sun.com/products/jms/.
[14] JSR 208: Java Business Integration (JBI). http://jcp.org/en/jsr/detail?id=208/.
[15] Tijs Rademakers and Jos Dirksen. Open-Source ESBs in Action. Manning Publications. September 2008.
[16] JSR 003: Java Management Extensions (JMX). http://jcp.org/aboutJava/communityprocess/final/jsr003/index3.html/.
[17] OSGi Alliance. http://www.osgi.org/.
[18] Michal Zaremba, Matthew Moran and Thomas Haselwanter. D13.4.v0.2. WSMX architecture. Technical report, June 2005. http://www.wsmo.org/TR/d13/d13.4/v0.2/
[19] Shafiq, O., Krummenacher, R., Martin-Recuerda, F., Ying Ding and Fensel, D. Triple Space Computing Middleware for Semantic Web Services. Proceedings of the 10th IEEE on International Enterprise Distributed Object Computing Conference Workshops, 2006.
[20] Erich Gamma, Richard Helm, Ralph Johnson and John M. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. ISBN 0-201-63361-2.
[21] John Domingue, Dieter Fensel and Rafael González-Cabero. SOA4All, Enabling the SOA Revolution on a Word Wide Scale. Proceeding of the 2nd IEEE International Conference on Semantic Computing, August 2008.
[22] Antonio J. Roa-Valverde, Jorge Martinez-Gil and José F. Aldana-Montes. Boosting Annotated Web Services in SAWSDL. International Symposium on Distributed Computing and Artificial Intelligence 2009. To appear.
[23] Gregor Hohpe and Bobby Woolf. Enterprise Integration Patterns : Designing, Building, and Deploying Messaging Solutions. ISBN 0321200683. Addison-Wesley, 2004.
[24] The OASIS Semantic Execution Environment Technical Commitee. Available at http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=semantic-ex/.
[25] Michael Stollberg and Barry Norton. A Refined Goal Model for Semantic Web Services. Second International Conference on Internet and Web Applications and Services (ICIW'07), pp.17.
[26] The OMG's CORBA website. Available at http://www.corba.org/.
[27] Ulrich Kster, Holger Lausen and Birgitta Knig-Ries. Evaluation of Semantic Service Discovery - A Survey and Directions for Future Research. Proceedings of the 2nd Workshop on Emerging Web Services Technology (WEWST07) in conjunction with the 5th IEEE European Conference on Web Services (ECOWS07), Halle (Saale), Germany , November 2007.
[28] Jinghai Rao and Xiaomeng Su. A Survey of Automated Web Service Composition Methods. Semantic Web Services and Web Process Composition. In LNCS, Vol. 3387/2005 (2005), pp. 43-54.
[29] OASIS Reference Model for Service Oriented Architecture 1.0. Available at http://docs.oasis-open.org/soa-rm/v1.0/soa-rm.pdf.
[30] T. Hasselwanter, P. Kotinurmi, M. Moran, T. Vitvar and M. Zaremba. WSMX: a Semantic Service Oriented Middleware for B2B Integration , In Proceedings of the 4th International Conference on Service Oriented Computing , Springer-Verlag LNCS series, December, 2006, Chicago, USA.
[31] Brahmananda Sapkota, Zhangbing Zhou, Omair Shafiq and Daniel Wutke. D4.5 Triple Space Integration with respect to WSMX. Technical Report, April 2009. http://www.tripcom.org/docs/del/D4.5.pdf

Antonio J. Roa-Valverde. This author became a member of the University of Málaga in 2007. He was born in 28st April 1984 in Puente Genil, Córdoba (Spain), studied Computer Science at the University of Málaga and is member of this University since the end of his degree. He is currently a Phd student and his research interests include SOA, EAI and Semantic Web Services.

José F. Aldana-Montes. This author became a member of the University of Málaga in 1990, and is full time professor since 1994. He was born in 20st November 1965 in Málaga, studied Computer Science at the University of Málaga. This author has published a lot of papers in the field of Data Bases, reasoners and Semantic Web.